Algoritmi di ordinamento

Sequential-sort, Bubble-sort, Quicksort

Definizione

- Dato un multi-insieme $V=\{V_n\}^{N-1}_{n=0}$ di valori in D, il problema dell'ordinamento è trovare una permutazione *n(j)* degli indici *n* tale che $j_2 \geq j_1 \Rightarrow V_{n(j2)} \geq V_{n(j1)}$

Sequential-sort

- Richiede la ricerca del massimo valore dell'insieme ed un'operazione di swap
- Si seleziona l'elemento massimo di V e si scambia di posizione con l'elemento nell'ultima posizione
  - si prosegue allo stesso modo sui primi N-1 elementi

Sequential-sort: costo

- $\Gamma_{SeqS}(N) = c_1 \cdot N + c_2 + \Gamma_{SeqS}(N-1)$ se $N>1$, c_2 se $N==0$

  dove $c_1 \cdot N$ costo di selezione, c_2 costo di swap
- Eseguendo i conti: $C_{SeqS}(N)=O(N^2)$

Sequential-sort: implementazione

```
void seqSort(struct data *V, int N, int *perm)
{
  int count, countMax, iter;

  for (count=0; count<N; count++)
    perm[count]=count;

  for (iter=0; iter<(N-1); iter++)
  {
    for (count=1, countMax=0; count<(N-iter); count++)
      if ( isGreater(V[perm[count]], V[perm[countMax]]) )
        countMax = count;
    swap( perm, countMax, N-iter-1 );
  }
}
```

Sequential-sort: ottimizzazione

- Si può cercare di sostituire massimo e minimo contemporaneamente
  - si dimezza il numero di iterazioni, che diventano più costose computazionalmente
  - la ricerca contemporanea di massimo e minimo è comunque meno costosa (il guadagno è qui)

Sequential-sort: ottimizzazione

- Ricerca contemporanea di massimo e minimo:

```
if (x>y)
{
  if (x>max)
    max=x;
  if (y<min)
    min=y;
} else {
  if (y>max)
    max=y;
  if (x<min)
    min=x;
}
```

NOTA: il costo rimane $O(N^2)$, stiamo riducendo i confronti da 4 a 3

Bubble-sort

- Richiede il confronto e lo swap di elementi successivi, iterando sui dati
- All'iterazione *k* si scandiscono le posizioni da 0 a N-*k*-1, confrontando gli elementi [posizione] e [posizione+1], eventualmente permutando le posizioni non ordinate
- Se non si scambia nessun elemento non si itera più

Bubble-sort: proprietà

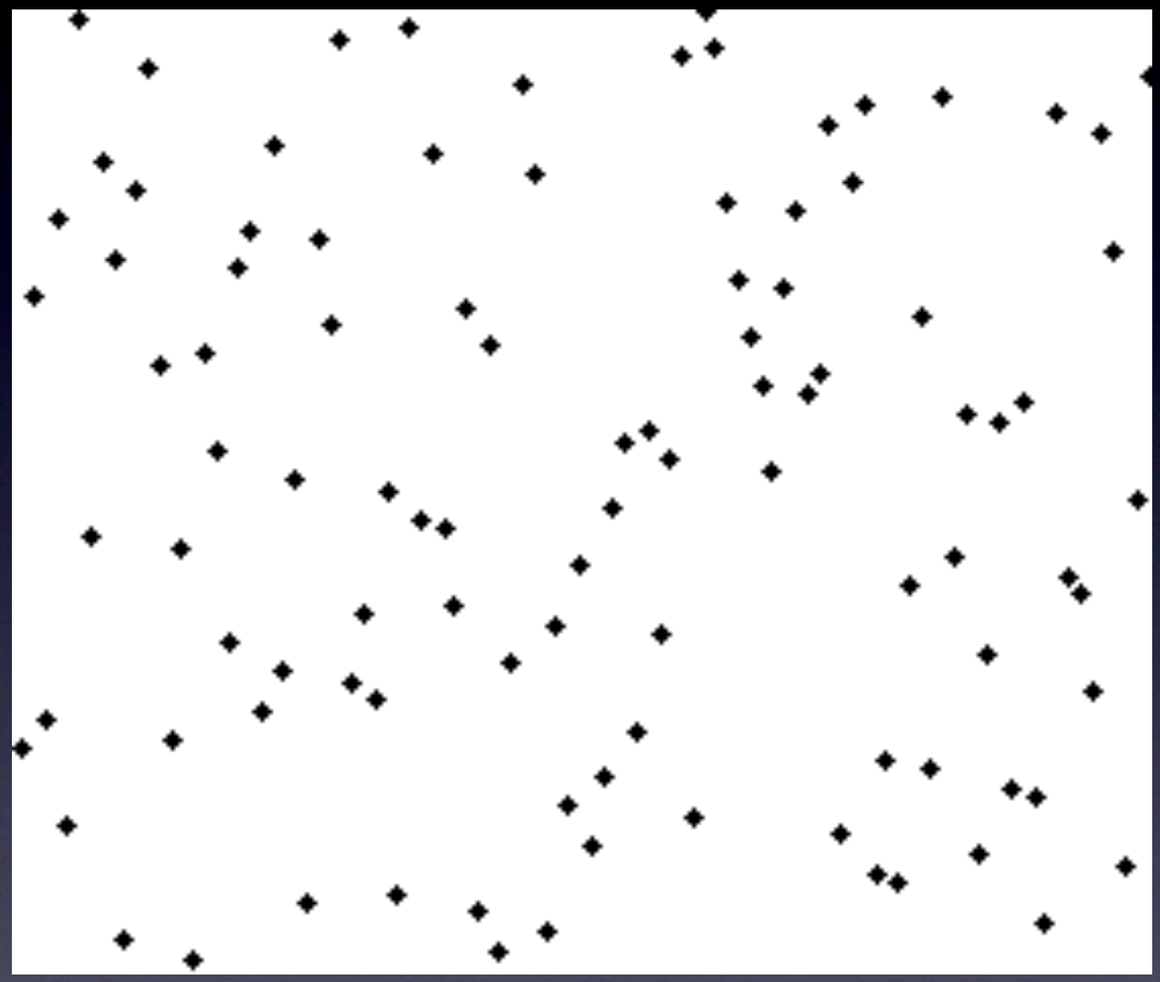
i. un elemento che all'inizio dell'iterazione NON è preceduto da alcun maggiorante avanza fino ad incontrare il primo; se non ha maggioranti avanza fino all'ultima posizione

ii. un elemento che all'inizio dell'iterazione è preceduto da almeno un maggiorante, al termine dell'iterazione è arretrato di UNA posizione

Bubble-sort: esempio

- L'algoritmo è asimmetrico: per le proprietà *i* e *ii* i valori più grandi salgono velocemente mentre i piccoli al massimo scendono di una posizione

Bubble-sort: esempio



- L'algoritmo è asimmetrico: per le proprietà *i* e *ii* i valori più grandi salgono velocemente mentre i piccoli al massimo scendono di una posizione

Bubble-sort: costo

- Dopo *k* iterazioni gli ultimi *k* elementi sono nelle posizioni corrette (proprietà *i*), dobbiamo iterare solo sulle prime N-*k* posizioni

- $\Gamma_{BubS}(N) = (s+c) \cdot N + \Gamma_{BubS}(N-1)$ se N>1, c_2 se N==0

  dove *c* è il costo di confronto, *s* costo di swap

- $\Gamma_{BubS}(N) = \frac{(s+c)(N-1)(N-2)}{2} \Rightarrow C_{BubS}(N) = O(N^2)$

Bubble-sort: costo

- Possiamo terminare senza aver fatto tutte le N-2 iterazioni: nell'algoritmo sequenziale NO
- Posso dover fare (N-1)(N-2) swap: nel sequenziale ne facciamo sempre (N-1)
  - il meccanismo di virtualizzazione dell'ordine diventa importante

Bubble-sort: implementazione

```
void bubbleSort(struct data *V, int N, int *perm)
{
  int iter;
  Boolean noSwap;
  Boolean swapFound;

  iter = 0;
  noSwap = FALSE;
  while ( noSwap == FALSE )
  {
    for (count=0, swapFound=FALSE; count<N-iter-1; count++)
    {
      if ( isSmaller(V[perm[count]], V[perm[count+1]]) )
      {
        swap( perm, count, count+1); // virtualizzazione dell'ordine
        swapFound = TRUE;
      }
    }
    if ( swapFound == FALSE )
      noSwap = TRUE;
    else
      iter++;
  }
}
```

Quicksort

- Algoritmo di tipo *divide et impera*
  - come Mergesort
  - introdotto nel 1960 da C.A.R. Hoare
  - tipica implementazione ricorsiva
  - disponibile nella libreria standard C
- Richiede di risolvere un problema di partizione

Quicksort: algoritmo di partizione

- Si riarrangiano gli elementi di un vettore in modo che un elemento pivot sia portato in posizione tale da avere alla sua sinistra solo elementi minori o uguali e a destra elementi maggiori
- Scelta del pivot: l'elemento in una posizione tipica del vettore (es. primo) o scelta casuale

Quicksort: algoritmo ricorsivo

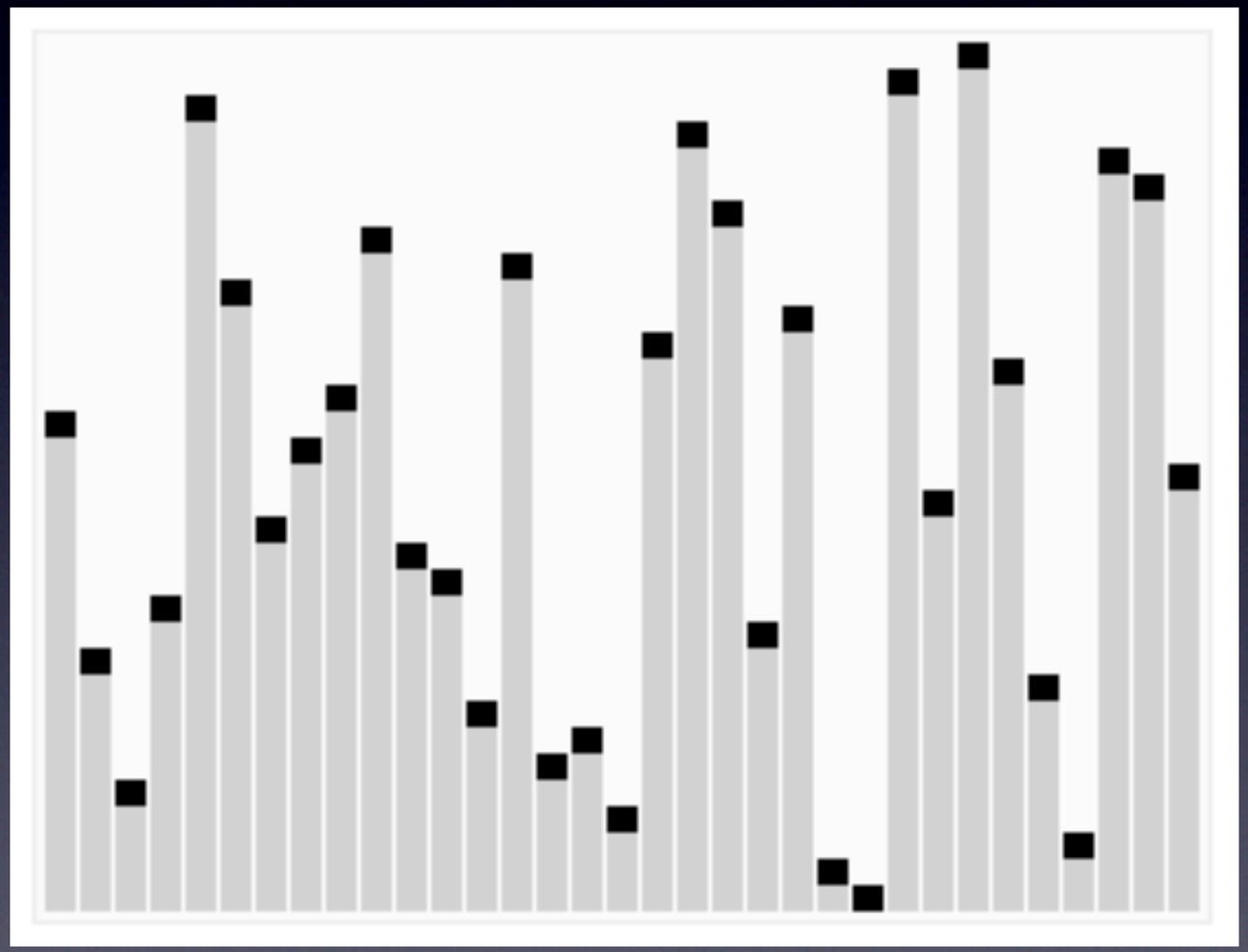
- si sceglie un elemento dell'array (il "pivot")
- si suddividono gli altri elementi in due gruppi:
  - "quelli piccoli" che sono minori del pivot
  - "quelli grandi" che sono maggiori o uguali al pivot
- si ordina ricorsivamente ogni gruppo

Quicksort: proprietà

- Gli elementi minori del pivot NON devono essere confrontati con i maggiori;
  altrettanto i maggiori con i minori
  - nell'ordinamento sequenziale o bubble-sort ogni elemento è confrontato con gli altri

Quicksort: esempio

Quicksort: esempio



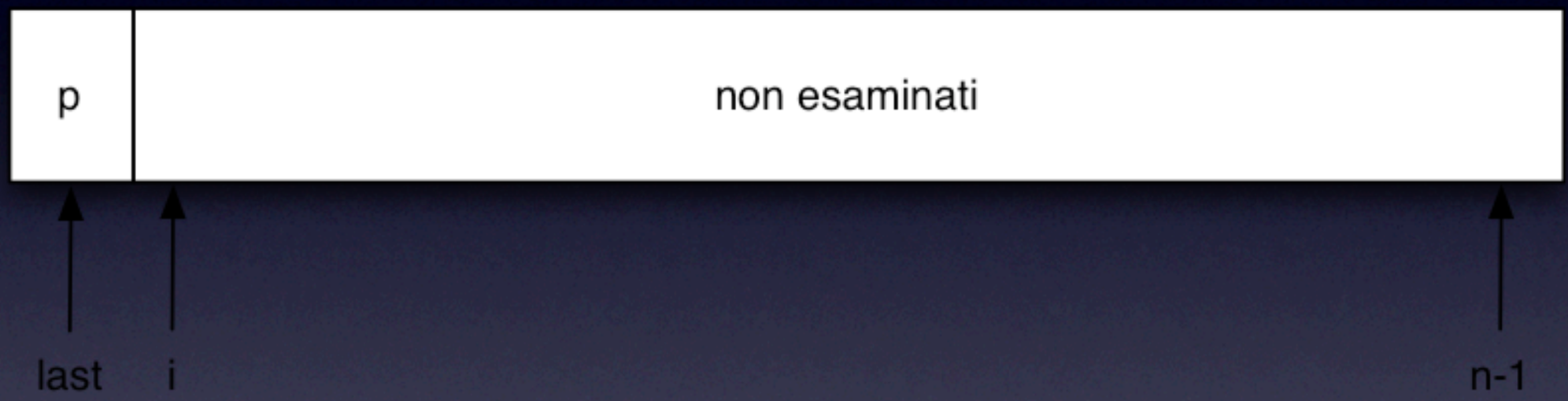
Quicksort: implementazione

```
void quicksort( int v[], int n)
{
  int i, last;

  if (n <= 1)
    return;
  swap(v, 0, rand()%n); // pivot random, spostato in v[0]
  last = 0;
  for ( i=1; i<n; i++ )
    if ( v[i] < v[0])
      swap( v, ++last, i);
  swap( v, 0, last ); // recupera il pivot
  // ordina ricorsivamente le due parti
  quicksort( v, last );
  quicksort( v+last+1, n-last-1 );
}
```

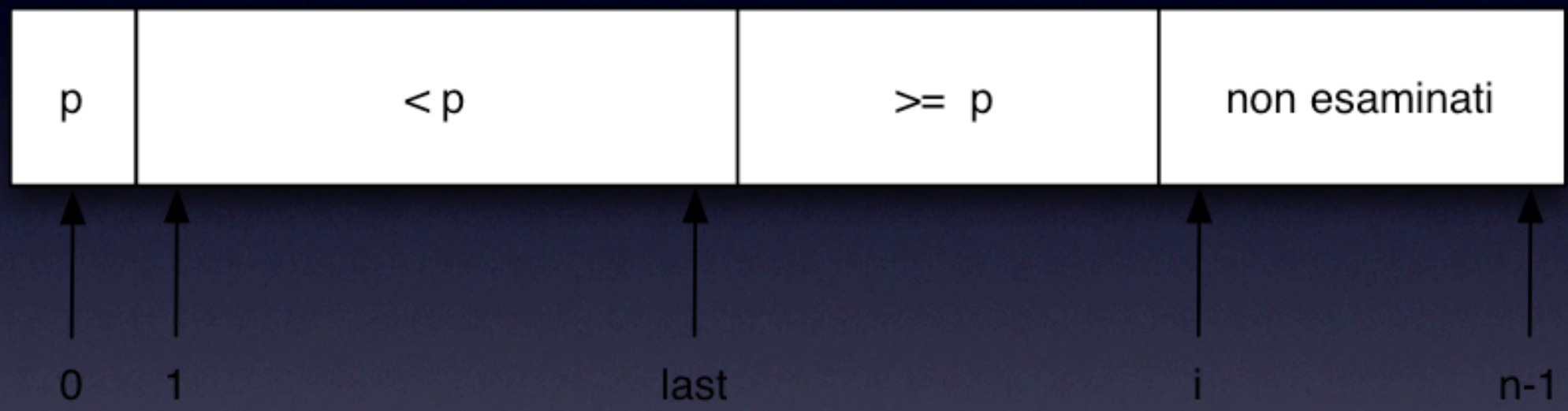
```
void swap(int v[], int i, int j)
{
  int temp;
  temp=v[i];
  v[i]=v[j];
  v[j]=temp;
}
```

- All'inizio del processo il pivot viene messo nella prima posizione



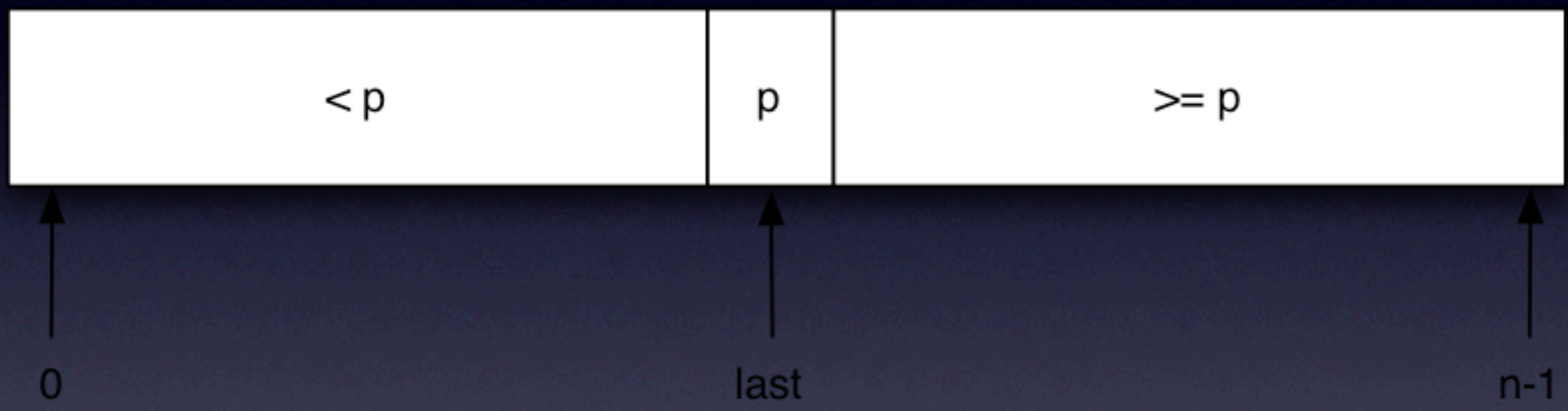
- Dopo che tutti gli elementi sono stati suddivisi l'elemento 0 viene scambiato con l'elemento last per rimettere il pivot nella posizione finale

- All'inizio del processo il pivot viene messo nella prima posizione



- Dopo che tutti gli elementi sono stati suddivisi l'elemento 0 viene scambiato con l'elemento last per rimettere il pivot nella posizione finale

- All'inizio del processo il pivot viene messo nella prima posizione



- Dopo che tutti gli elementi sono stati suddivisi l'elemento 0 viene scambiato con l'elemento last per rimettere il pivot nella posizione finale

Quicksort: funzione di libreria

- La funzione di libreria qsort deve ricevere la funzione di comparazione come parametro
  - si usa un puntatore a funzione
  - la funzione deve poter lavorare su qualsiasi tipo, quindi riceve puntatori void e poi ne fa il cast appropriato

Quicksort: funzione di libreria

- void qsort
  (void *base, size_t nel, size_t width, int
  (*compar)(const void *, const void *));

Quicksort di *nel* oggetti, il cui elemento iniziale è puntato da *base*. Ogni oggetto ha dimensione *width*. La comparazione è fatta dalla funzione *compar*.

Funzione di comparazione: esempio

```
int icmp(const void *p1, const void *p2)
{
   int v1, v2;
   v1 = *(int *)p1;
   v2 = *(int *)p2;
   if( v1 < v2 );
      return -1;
   else if ( v1 == v2 )
      return 0;
   else
      return 1;
}
```

Quicksort: esempio di uso della libreria

```
int arr[N];

qsort( arr, N, sizeof(arr[0]), icmp);
```

Esempio di uso di puntatore a funzione

```
     #include <stdio.h>
#include <stdlib.h>
#include <string.h>

void controlla(char *a, char *b, int (*cmp)(char *a1, char *b1));


int main (int argc, const char * argv[]) {
    char s1[80],s2[80];
   int (*p)();
   p=strcmp;

   gets(s1);
   gets(s2);

   controlla(s1,s2,p);

    return 0;
}

void controlla(char *a, char *b, int (*cmp)(char *a1, char *b1))
{

   printf("Controllo %s e %s\n",a,b);
   if(!(*cmp)(a,b))
          printf("Uguali\n");
   else
          printf("Diversi\n");
}
```